Functional Overlap Reranking for Neural Code Generation

Hung Quoc To , Minh Huynh Nguyen , Nghi D. Q. Bui
FPT Software AI Center, Viet Nam
hungtq29@fpt.com, minh.nghminh@gmail.com, bdqnghi@gmail.com

Abstract

Code Large Language Models (CodeLLMs) have ushered in a new era in code generation advancements. However, selecting the best code solutions from all possible CodeLLM outputs remains a challenge. Previous methods often overlooked the intricate functional similarities and interactions between solution clusters. We introduce ***SRank***, a novel reranking strategy for selecting the best solutions from code generation, focusing on modeling the relationships between clusters of solutions. By quantifying the functional overlap between solution clusters, our approach provides a better ranking strategy for code solutions. Empirical results show that our method achieves remarkable results on the pass@1 score. For instance, on the Human-Eval benchmark, we achieve 69.66% in pass@1 with Codex002, 75.31% with WizardCoder, 53.99% with StarCoder, and 60.55% with CodeGen, surpassing state-of-the-art code generation reranking methods such as CodeT and Coder-Reviewer on the same CodeLLM by a significant margin ($\approx 6.1\%$ **improvement on average**). Even in scenarios with a limited number of sampled solutions and test cases, our approach demonstrates robustness and superiority, marking a new benchmark in code generation reranking. Our implementation can be found at https://github.com/FSoft-AI4Code/SRank-CodeRanker.

1 Introduction

Recent advancements in language models tailored for code, known as Code Large Language Models (CodeLLMs) (Luo et al., 2023; Wang et al., 2023; Nijkamp et al., 2023b; Rozière et al., 2023; Wei et al., 2023; Lozhkov et al., 2024; Pinnaparaju et al., 2024; Guo et al., 2024; Bui et al., 2023), have garnered significant interest, particularly due to the expansion of large-scale language models and the volume of pre-training data (Kaplan et al., 2020; Zhao et al., 2023). A primary utility of CodeLLMs is their capacity to generate code given natural language descriptions written by humans (Chen et al., 2021; Fried et al., 2023; Chowdhery et al., 2022; Nijkamp et al., 2023b). However, prior studies (Holtzman et al., 2020; Austin et al., 2021) have highlighted that the sequences generated by these models can be prone to errors, especially when likelihood-based decoding techniques like greedy search and beam search are employed. Alternatively, sampling-based decoding techniques (Fan et al., 2018; Holtzman et al., 2020) extract multiple solutions from the model's multinomial distribution. This method generates a wide range of code solutions, many of which are correct (Austin et al., 2021; Ni et al., 2023). As a result, there is a growing interest in developing reranking strategies for code generation (Li et al., 2022; Inala et al., 2022; Chen et al., 2023; Zhang et al., 2023; Ni et al., 2023), with the goal of sorting through an abundance of sampled solutions to identify high-quality and accurate ones.

The goal of reranking is to organize the set of candidate programs so that accurate programs are prioritized. Li et al. (2022), Chen et al. (2023), and Ni et al. (2023) have clustered code solutions based on their functionality, then used cluster-specific data to determine ranking scores. Given that language models frequently produce code solutions that differ syntactically but are semantically analogous, functional clustering narrows the candidate pool. The emphasis then shifts from ranking individual solutions to ranking clusters themselves. Previous ranking strategies, such as AlphaCode (Li et al., 2022) and CodeT (Chen et al., 2023), provide approaches to clustering and reranking code solutions. While AlphaCode (Li et al., 2022) focuses on identical outputs from model-generated test inputs, CodeT (Chen et al., 2023) focuses on solutions that pass model-generated test cases. The Coder-Reviewer approach (Zhang et al., 2023), inspired by collaborative software development, uses a dual-model system to cross-check generated programs

Findings of the Association for Computational Linguistics: ACL 2024, pages 3686–3704
August 11-16, 2024 ©2024 Association for Computational Linguistics

against language instructions. However, by treating clusters in isolation, they fail to model potentially informative functional similarities and interactions across clusters.

To address this limitation, we propose **SRank**, a novel reranking approach emphasizing *modeling inter-cluster* relationships. Specifically, we introduce a new metric called *functional overlap* to quantify the similarity between clusters based on their execution outputs. This allows for identifying the most representative cluster that exhibits maximal overlap with all other clusters. As inconsistencies often indicate incorrect functionality, the cluster interacting most comprehensively likely represents the optimal solution. By incorporating these inter-cluster relationships into the ranking pipeline, we can better identify the most promising solutions. Through extensive evaluation, we demonstrate that modeling inter-cluster relationships and functional overlap provides significant and consistent improvements over prior state-of-the-art solution ranking methods (Li et al., 2022; Chen et al., 2023; Zhang et al., 2023) on a wide range of state-of-the-art CodeLLMs, including Codex, WizardCoder, StarCoder, and CodeGen. For instance, on the HumanEval benchmark, our method achieved a pass@1 score of 75.31% with WizardCoder34B, outperforming the Coder-Reviewer's score of 66.9%. Similarly, on the MBPP-S benchmark, our method improved the pass@1 score for WizardCoder from 50.3% with Coder-Reviewer to 51.03% with our approach. Similar improvements are applied for other CodeLLMs, including StarCoder, CodeGen, and Codex002. If we compare SRankwith a simple random sampling method to get code solutions, we observe massive improvements across the models, with average improvements of 23.07% and 17.64% for HumanEval and MBPP-S, respectively. Our evaluation is more *comprehensive* because we include many SOTA CodeLLMs of varying sizes, whereas CodeT and Coder-Reviewer did not. This provides compelling evidence of our approach's robustness across a wide range of models.

We also conducted an extensive analysis to demonstrate some of our advantages, such as our approach's remarkable robustness even with limited solutions and test cases. In summary, by moving from isolated clusters to interacting clusters with quantified functional overlap, our novel reranking strategy aims to address the limitations of prior ranking techniques for code generation. To summarize our contributions, they are as follows:

- We introduce a novel reranking strategy for CodeLLMs that emphasizes the inter-cluster relationships and leverages the functional overlap between them, providing a more robust and accurate approach to pick the best solutions.

- Through extensive and comprehensive evaluations, we demonstrate that our approach consistently outperforms existing state-of-the-art methods in code generation. For instance, our method achieved superior results on both the HumanEval and MBPP-S benchmarks across various CodeLLMs.

- We perform extensive analysis to evaluate the robustness of our method, highlighting its effectiveness even with a limited number of sampled solutions and test cases, and its ability to capture intricate interactions between clusters, setting it apart from previous ranking techniques.

2 Background & Motivation

2.1 Code Generation

Code generation involves generating code solutions to programming problems based on a given context c. The context includes a natural language description and a code snippet containing statements such as imports and a function signature. Additionally, a predefined set of test cases, denoted as T, is provided to evaluate the correctness of the generated code solutions. Using c as the input on CodeLLM, we obtain a set of solutions $\mathbf{S} = \{s_1, s_2, ..., s_N\}$, where N is a hyperparameter defining the number of return sequences from the CodeLLM execution. A solution s is considered valid if it successfully passes the predefined set of test cases $\mathbf{T}$.

2.2 Solution Clustering and Reranking

The reranking task aims to prioritize correct programs in the candidate list $\mathbf{S}$. Previously, solutions were clustered by functionality, simplifying the task due to the language models' tendency to produce syntactically varied but semantically similar solutions. Thus, the focus shifts from ranking individual solutions to ranking these functional clusters.

For instance, AlphaCode (Li et al., 2022) uses a distinct model to create test inputs. Solutions are then executed against these inputs, and those with

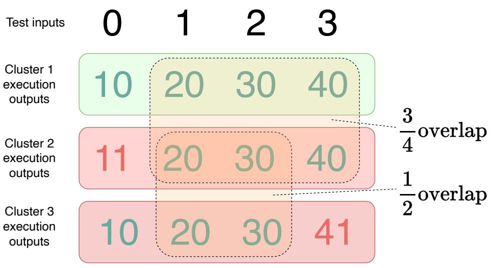

Test inputs
0
1
2
3
Cluster 1 execution outputs
10
20
30
40
Cluster 2 execution outputs
11
20
30
40
Cluster 3 execution outputs
10
20
30
41
$\frac{3}{4}$overlap
$\frac{1}{2}$overlap

Figure 1: Illustration on concept of "functional overlap" among clusters of solutions. Cluster 1 outputs [10,20,30,40]. Cluster 2's output is [11,20,30,40]. Cluster 3's output is [10,20,30,40]. As a result, Cluster 1 overlaps Cluster 2 on three values [20,30,40], indicating that they are 3/4 overlapped. Cluster 1 overlaps Cluster 3 on three values [10,20,30], which can also be considered 3/4 overlapped. Cluster 1 has a functional overlapping score of 3 + 3 = 6. Cluster 2 overlaps with Cluster 3 on two values [20,30], resulting in a functional overlapping score of 2 + 3 = 5, and Cluster 3 has a functional overlapping score of 5. Thus, Cluster 1 has the highest cumulative functional overlap, is most representative and likely to be the optimal solution.

matching outputs are clustered. The reranking strategy is based on the understanding that while there can be numerous incorrect program variations, correct ones often show similar patterns, leading to clusters being ranked by their solution count.

Conversely, CodeT (Chen et al., 2023) clusters solutions that pass the same model-generated test cases. However, this can group functionally diverse solutions. If a cluster's solutions only pass some test cases, it's uncertain if the outputs for the failed cases are consistent across solutions, potentially compromising cluster functionality and the confidence in selecting from these ranked clusters. We show a concrete example to analyze this problem in Appendix G.

2.3 Modeling Inter-Cluster Relationships

Existing clustering and reranking methods analyze clusters independently without considering inter-cluster relationships (Li et al., 2022; Chen et al., 2023). However, modeling these interactions can better indicate cluster correctness. As such, we propose a new metric called *"functional overlap"* to quantify cluster similarity based on execution outputs, as shown in Figure 1. We can execute code solutions from each cluster on the same test inputs and compare their outputs. The level of output match indicates the extent of functional overlap between two clusters.

The intuition is that clusters with high overlap exhibit greater shared functionality. By modeling the extent to which a cluster overlaps with others, functional overlap identifies the most "representative" cluster. A cluster with maximal cumulative overlap has outputs most consistent with all other clusters. As inconsistencies often indicate incorrect functionality, the cluster interacting most comprehensively is likely the optimal solution. This is similar to the assumptions of Fischler and Bolles (1981), where incorrect solutions are diverse and there is a low probability of having a functional agreement among incorrect solutions.

3 Approach Details

3.1 Overview

Figure 2 provides an overview of our end-to-end approach. First, given a well-trained CodeLLM, e.g., Codex, and three inputs: (1) Task description, (2) Code generation prompt, (3) Test case generation prompt, we instruct the CodeLLM to generate a set of code solutions as well as test cases. Specifically, we prompt the CodeLLM to produce a collection of code solutions $\mathbf{S} = \{s_1, s_2, ..., s_N\}$ and a set of test cases $\mathbf{T} = \{t_1, t_2, ..., t_M\}$, where N and M are hyperparameters defining the number of solutions and test cases.

Each test case t_i consists of two components: the test input z_i and the expected output $\hat{o}_i$ based on the context (e.g., `assert add(1,2) == 3`, where `(1,2)` is the input and `3` is the output). We can then execute the test inputs $\mathbf{Z} = \{z_1, z_2, ..., z_M\}$ on the set of solutions $\mathbf{S}$ to generate the execution outputs $\mathbf{O} = \{o_{11}, o_{12}, ..., o_{NM}\}$. Next, we cluster the solutions $\mathbf{S}$ into groups $\mathbf{C} = \{C_1, C_2, ..., C_K\}$ based on their execution outputs, where K is the number of unique clusters. We then compute an interaction matrix $\mathbf{I}$ to quantify the functional overlap between clusters. Finally, we multiply the interaction matrix $\mathbf{I}$ by a validation score vector $\mathbf{V}$ to obtain final ranking scores $\mathbf{R}$ for selecting the optimal solutions. The validation scores in $\mathbf{V}$ represent features of each cluster, such as the number of solutions.

In the following sections, we elaborate on the key steps of our algorithm.

3.2 Clustering Solutions by Execution Outputs

We first execute each solution $s_i \in \mathbf{S}$ on the test inputs $\mathbf{Z}$ to produce execution outputs $\mathbf{O}$. Solutions that exhibit identical execution outputs are grouped

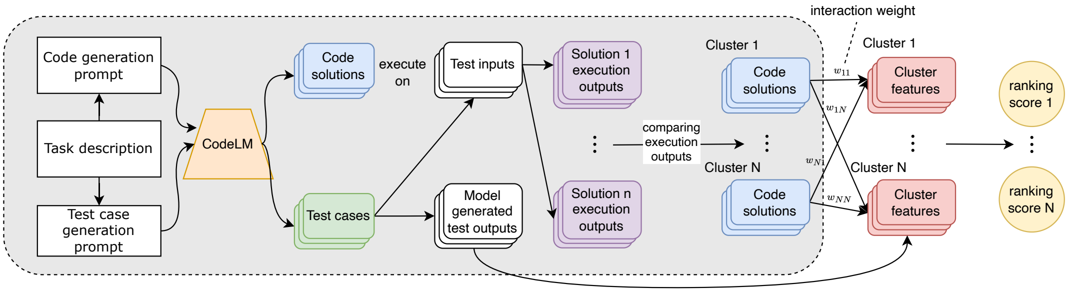


Figure 2: Method overview.

into the same cluster:

$$F(s_i) = F(s_j) \iff \mathbf{O}_{s_i} = \mathbf{O}_{s_j}.$$

Here, F represents the clustering function that maps a solution s to a cluster identifier C_k. The above equation indicates that two solutions s_i and s_j are assigned to the same cluster if and only if their output sets $\mathbf{O}_{s_i}$ and $\mathbf{O}_{s_j}$ are exactly equal.

3.3 Computing Interaction Matrix

After obtaining execution outputs o_{ij} for each cluster C_i on test input z_j, we define an interaction matrix $\mathbf{I} \in \mathbb{R}^{K \times K}$ to quantify functional overlap:

$$I_{ij} = \frac{1}{M} \sum_{k=1}^{M} \delta(o_{ik} = o_{jk}), \tag{1}$$

where o_{ik} and o_{jk} refer directly to the execution outputs of clusters C_i and C_j, respectively, on the k^{th} test input. δ is the indicator function that returns 1 if the condition inside is true and 0 otherwise.

3.4 Computing Final Ranking Scores

In addition to modeling inter-cluster interactions via $\mathbf{I}$, we also consider an extra validation dimension $\mathbf{V} \in \mathbb{R}^{K \times 1}$ containing cluster features. For instance, V_i could represent the number of solutions in cluster C_i (abbreviated as cluster sizes) or the number of test cases that the solutions in cluster C_i passed (abbreviated as pass rates), providing a notion of cluster confidence. The final ranking vector $\mathbf{R} \in \mathbb{R}^{K \times 1}$ can then be computed as $\mathbf{R} = \mathbf{I} \cdot \mathbf{V}$. Here, R_i aggregates information about both the inter-cluster interactions of C_i (via $\mathbf{I}$) and its cluster features (via $\mathbf{V}$). Clusters with higher ranking scores in $\mathbf{R}$ are those with significant functional overlap to other clusters and high validity according to $\mathbf{V}$. By considering inter-cluster relationships and functional similarity in a principled manner, we believe our ranking approach can effectively identify the most promising solutions. We validate our method through extensive experiments in the following sections.

4 Experimental Setup

Models We evaluate our method on several state-of-the-art CodeLLMs, including Codex, WizardCoder, StarCoder, and CodeGen. Each model family has different model sizes (e.g., WizardCoder 15B and 34B) and different training methods (e.g., base model and instruction fine-tuned model). As a result, we chose a diverse set of models, ranging from small to large scale, and from base models to instruction fine-tuned models, to demonstrate the efficacy of our ranking method. In total, we demonstrate our approach on 6 models ranging from 6B to 34B parameters.

Metrics We use pass@k (Chen et al., 2021), which is often employed to evaluate the functional correctness of code solutions based on code execution instead of similarity-based metrics.

Baselines We compare SRank with recent methods for solution reranking, including Coder-Reviewer (Zhang et al., 2023) and CodeT (Chen et al., 2023). Coder-Reviewer (Zhang et al., 2023) is the state-of-the-art method. On the other hand, CodeT (Chen et al., 2023) shares a similar clustering-reranking approach to our method.

Benchmarks We use two popular benchmarks in code generation: HumanEval (Chen et al., 2021) and MBPP-S (sanitized version) (Austin et al., 2021). For a more challenging assessment, we evaluate **SRank** on APPS (Hendrycks et al., 2021). To avoid exposing real test cases to the language model, we follow the prompt design in (Chen et al., 2021) by removing all example input-output cases from context before generating code solutions and test cases.

Implementation Details For Codex002 and CodeGen16B, we refer to the artifacts, including both solutions and test cases, provided by Chen et al. (2023). Regarding the remaining models, we use the HuggingFace library (Wolf et al., 2019) and load models in half-precision. We set the temperature to 0.8, the top p to 0.95, the maximum new tokens to 2048, and the timeout for executing solutions to 5 seconds. For each problem, we sample 100 code solutions and 100 sequences of test cases, each sequence containing multiple test cases. The prompts we used for sampling code solutions and test cases from each model can be found in Appendix B. For post-processing code solutions and test cases, we follow Chen et al. (2023) to truncate the generated sequences by the five-stop words: "\nclass", "\ndef", "\n#", "\nif", and "\nprint".

5 Experimental Results

Table 1 presents the pass@1 results on the HumanEval and MBPP-S benchmarks for various CodeLLMs. Our method, SRank, consistently outperforms other techniques across most models and benchmarks. For instance, on the HumanEval benchmark, SRank achieves average improvements over CodeT and Coder-Reviewer of about 3.63% and 8.81% in pass@1, respectively.

Additionally, when comparing Coder-Reviewer with the random sampling method, it is unstable across the models. Specifically, using WizardCoder15B and StarCoder as examples, Coder-Reviewer brings modest increases of 4.17% and 6.16%, compared to our improvements of 14.79% and 21.44%. On the MBPP-S benchmark, **SRank** still achieves outstanding performance, although the magnitude of improvements is slightly less than that of HumanEval. Our comprehensive experiments demonstrate the effectiveness of **SRank** over CodeT and Coder-Reviewer.

To assess the effectiveness of our proposed method on a wide range of coding problem difficulties, we evaluate **SRank** on APPS using the Codex002. Results are shown in Table 3. We observe that **SRank** consistently outperforms all other baselines by a significant margin. Compared with CodeT, adding inter-cluster modeling significantly improves reranking results. On the other hand, due to the difficulty of the tasks, the improvement is less significant when the level of difficulty increases. The results show that **SRank** is robust and scales well with different difficulty levels.

In addition, to validate our method on closed source and high capability models, we select Anthropic Claude 3 Opus (Anthropic, 2024) as a representative LLM in our experiment. Please refer to Appendix F for results.

6 Analysis

Assumption Validation We provide a comprehensive analysis to validate our assumption that *incorrect solutions are diverse and there is a low probability of functional agreement among incorrect solutions*. Formally, let $\mathbf{S}$ denote the set of solutions sampled from a certain CodeLLM, s_i be the i-th solution in $\mathbf{S}$, $\mathbf{C_k}$ be the k-th cluster by our clustering algorithm, C^i be the cluster including the solution s_i, and $|\mathbf{C}_k|$ and $|\mathbf{C}_k|^*$ be the number of solutions and quantity of incorrect solutions in the k-th cluster, respectively. The function $f(s_i, s_j)$ is defined as the computation of the functional overlap between s_i and s_j, similar to Eq. 1. We then compute the probability of incorrect solutions with varying levels of functional overlap.

$$p(\mathrm{l} \leq f(s_i, s_j) < \mathrm{h}, s_i \text{ and } s_j \text{ are incorrect}) = \frac{\sum_{(i,j)\in\mathcal{M}} |C^i|^* |C^j|^*}{\binom{|S|}{2}} \quad (2)$$

Here, $\mathcal{M}$ is comprised of pairs of (s_i, s_j) where $\mathrm{l} \leq f(s_i, s_j) < \mathrm{h}$, and l and h are the two hyperparameters. We consider two range values, $(\mathrm{l}_1, \mathrm{h}_1)$ and $(\mathrm{l}_2, \mathrm{h}_2)$ with the same length, where $\mathrm{l}_1 < \mathrm{l}_2$ and $\mathrm{h}_1 < \mathrm{h}_2$. According to our assumption, we anticipate that the following inequality holds:

$$p(\mathrm{l}_1 \leq f(s_i, s_j) < \mathrm{h}_1, s_i \text{ and } s_j \text{ are incorrect}) > p(\mathrm{l}_2 \leq f(s_i, s_j) < \mathrm{h}_2, s_i \text{ and } s_j \text{ are incorrect})$$

The left term denotes the probability of lower functional agreement among incorrect solutions, while the right term signifies the corresponding probability of higher functional agreement among incorrect solutions. The observed relationship indicates that the probability of lower functional agreement surpasses that of higher functional agreement, substantiating our assumption. Results in Figure 3 show a general decline in probability with increasing values of l and h. Particularly for the range $(\mathrm{l}, \mathrm{h}) = (0, 0.1)$, the probability is significantly higher compared to those of other ranges, and at the next range, $(\mathrm{l}, \mathrm{h}) = (0.1, 0.2)$, the probability experiences a notable decrease. Moreover, when incorrect solutions exhibit a high functional overlap, exceeding 0.7, the probability is low at around 3%. These findings are consistent with our assumption. More results can be found in Appendix A.

	HumanEval					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Greedy	68.90	50.61	28.05	39.63	47.00	29.70
CodeT	72.36	58.64	56.81	50.51	65.80	36.70
Coder-Reviewer	-	49.37	45.63	38.71	66.90	42.60
Random	59.88	45.20	26.68	32.55	37.06	22.78
SRank	**75.31**	**59.99**	**60.55**	**53.99**	**69.66**	**43.07**
	MBPP-S					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Greedy	60.42	51.29	42.86	45.90	58.10	42.40
CodeT	63.39	58.18	55.02	58.05	67.70	49.50
Coder-Reviewer	-	52.52	52.74	49.48	64.70	50.30
Random	54.37	45.72	34.60	39.26	47.50	31.54
SRank	**64.14**	**59.01**	**57.02**	**58.38**	**69.25**	**51.03**

Table 1: Results of pass@1 on HumanEval and MBPP-S benchmarks in the zero-shot setting compared to SOTA methods, CodeT and Coder-Reviewer.

	HumanEval					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Cluster sizes	72.17	56.38	55.92	48.63	59.43	40.51
Pass rates	65.09	43.07	36.17	35.28	58.37	21.89
Cluster sizes + Pass rates	73.28	58.21	58.35	51.90	66.07	41.72
Interaction + Cluster sizes	73.79	58.16	59.46	53.12	65.84	42.48
Interaction + Pass rates	73.59	53.49	48.37	51.13	65.91	34.61
SRank (all)	**75.31**	**59.99**	**60.55**	**53.99**	**69.66**	**43.07**
	MBPP-S					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Cluster sizes	65.46	56.17	56.76	55.50	64.22	53.00
Pass rates	61.88	49.93	43.80	48.57	60.80	36.67
Cluster sizes + Pass rates	64.38	58.13	**57.30**	57.68	68.78	50.60
Interaction + Cluster sizes	**66.39**	56.80	55.61	55.58	66.50	**53.08**
Interaction + Pass rates	63.33	56.11	50.27	51.33	64.53	42.78
SRank (all)	64.14	**59.01**	57.02	**58.38**	**69.25**	51.03

Table 2: Results of Ablation Study on combining different cluster features

Method	Introduction	Interview	Competition
Random	20.35	3.11	0.74
Greedy	27.20	5.10	1.80
CodeT	34.60	8.10	2.20
SRank	**37.79**	**9.53**	**3.29**

Table 3: Results of pass@1 on APPS benchmakr using Codex002 model in the zero-shot setting compared baselines.

Impact of Cluster Features We aim to evaluate reranking performance solely with cluster features, i.e., only cluster sizes, pass rates, or a combination of both. Table 2 shows SRank's performance when these features are added or removed from the ranking pipeline. When ranking code solutions using only one of the mentioned features, we can see that *cluster sizes* are more important than *pass rates*. However, this does not mean that *pass rates* are unimportant. When both features are combined, the results can be improved even further. Finally, we achieve the best SRank performance by combining both of these features or one of them with the Interaction matrix.

It's worth noting that the cluster features possess a general and adaptable nature, allowing for potential extensions to incorporate additional information such as likelihood. We also report results of the combination of clusters' likelihood and Coder-Reviewer ranking criteria as cluster features with functional overlap in Appendix D.

Impact of Interaction Matrix We conduct additional experiments to demonstrate the effectiveness of the interaction matrix **I**. From the results in Table 2, it is obvious that the interaction matrix

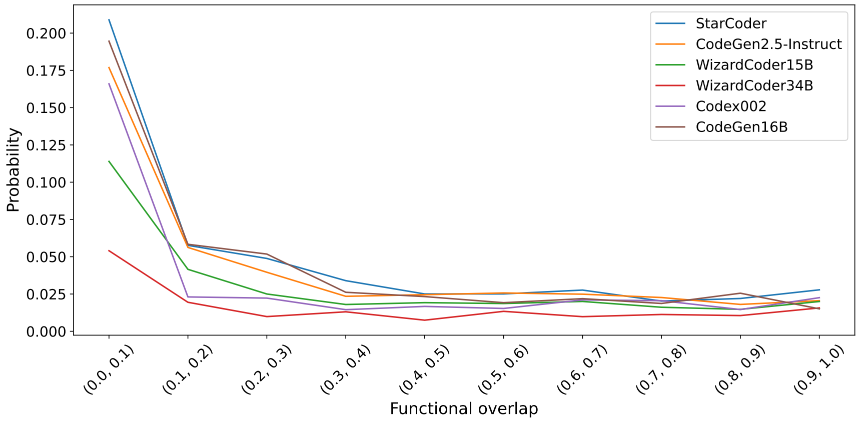


Figure 3: Probability of incorrect solutions varied based on the degree of functional agreement on HumanEval.

I helps to boost performance for any cluster features. Importantly, integrating *cluster sizes* with **I** achieves results on par with CodeT, highlighting the significance of interactions among clusters.

Scaling Number of Generated Test Cases We conducted an ablation study to assess how the number of generated test cases influences code reranking performance. Figure 4 shows the pass@1 performance as a function of the number of test cases generated, ranging from 2 to 50, on the HumanEval benchmark. Please refer to Appendix C for the results of MBPP-S.

Comparing each pair of solid line (with interaction matrix) and dashed line (w/o interaction matrix), cluster interaction consistently enhances performance over solely ranking by cluster features. The performance gap of reranking with and without interaction increases as the number of generated test cases increases, showcasing our method's scalability. However, with limited test cases, SRanksometimes underperforms due to the potentially negative impact of the cluster features when integrated with cluster interaction. For an optimal balance between effectiveness and efficiency, we suggest generating at least 30 test cases to fully benefit from cluster interaction, feasible within 1 to 2 sampling rounds given many test cases are generated within a single sampled sequence.

Additionally, comparing the performance of our method and CodeT reveals disparities in some CodeLLMs. This discrepancy arises from the clustering quality of CodeT, such that limited, low-quality test cases can lead to semantically inconsistent clusters, making ranking them less meaningful. In contrast, our method clusters solutions by matching execution outputs, which ensures functional consistency, enhancing ranking reliability.

Scaling Number of Sampled Solutions In a real-world setting, it is practical to have a limit on the number of both solutions and test cases generated, since sampling multiple sequences from LLM is costly and time inefficient for low resource computing. In this section, we examine the efficiency of our SRankby scaling down the number of sampled solutions to be less than or equal to 50 samples.

For practical reasons, we only run each experiment with 50 test cases generated by each model, rather than all test cases extracted from 100 sequences of test cases. We prompt WizardCoder34B, WizardCoder15B, and CodeGen2.5-Instruct to generate 50 different test cases in a single sequence. This prompting technique improves the overall pipeline efficiency by significantly lowering the computational overhead of sampling from CodeLLM. Figure 5 shows the pass@1 performance with the number of sampled solutions ranging from 2 to 50. Results of the MBPP-S benchmark can be found in Appendix C. The figure shows a similar trend to when we scale the number of generated test cases. Indeed, adding cluster interaction along with cluster features brings a performance boost, even when the number of sampled code solutions is small.

Compared to CodeT, as explained earlier, due to the semantically consistent clustering process in our method, both reranking with or without interaction outperform CodeT.

Moreover, with a limited number of sampled solutions and test cases, the results bypass greedy search (represented by the black lines). This proves both the effectiveness and the efficiency of our approach.

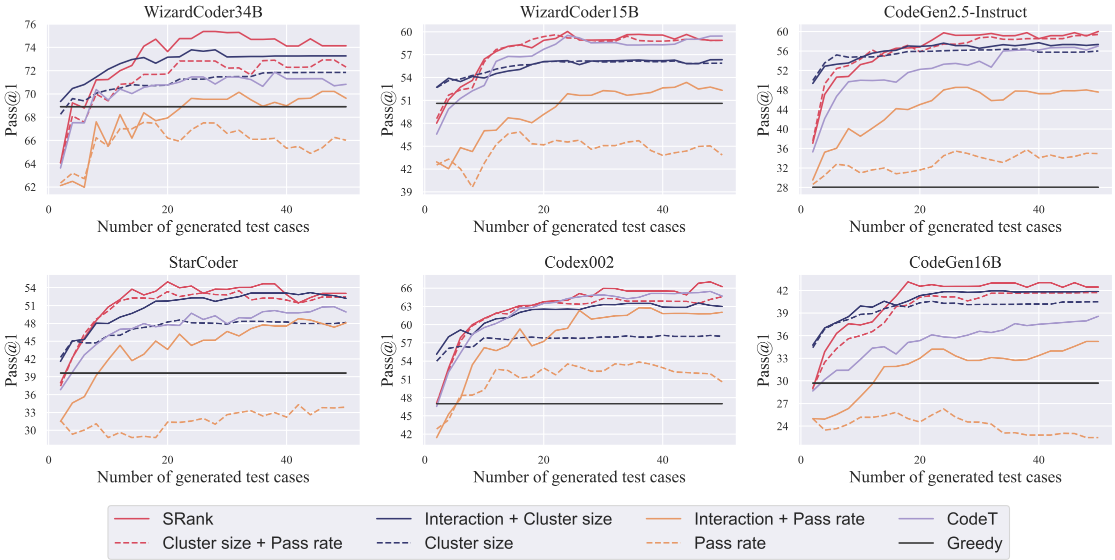


Figure 4: Ablation study on scaling number of model generated test cases vs. pass@1 on HumanEval.

7 Related Work

Code Large Language Models The emergence of large language models (LLMs) has transformed code understanding and generation tasks (Rozière et al., 2023; Li et al., 2023a; Nijkamp et al., 2023a; Wang et al., 2023; Nijkamp et al., 2023b; Fried et al., 2023; Li et al., 2023b). Recent research has employed natural language processing models for code-related tasks, using pretraining strategies akin to those for natural languages (Feng et al., 2020; Wang et al., 2021; Guo et al., 2020; Ahmad et al., 2021; Elnaggar et al., 2021; Peng et al., 2021; Kanade et al., 2020; Chakraborty et al., 2022; Ahmed and Devanbu, 2022; Niu et al., 2022). Among various CodeLLMs, those with larger capacities tend to perform better in code generation tasks. For instance, StarCoder (Li et al., 2023a) and CodeLlama (Rozière et al., 2023) handle contexts with up to 8,000 and 100,000 tokens, respectively, while WizardCoder (Luo et al., 2023) excels in evolving instructions. Additionally, models like CodeGen2.5-Instruct (Nijkamp et al., 2023a), CodeT5+Instruct (Wang et al., 2023), Codex002 (Chen et al., 2021), CodeGen-Mono 16B (Nijkamp et al., 2023b), and InCoder 6B (Fried et al., 2023) have also shown promise in this domain.

Reranking Methods for Code Generation Several studies have explored reranking code generated by language models (Chen et al., 2021; Zhang et al., 2023; Ni et al., 2023; Chen et al., 2023; Inala et al., 2022; To et al.), prioritizing solutions from these models. Chen et al. (2021) showed empirically that selecting solutions based on the mean log probability of tokens improves performance. Coder-Reviewer (Zhang et al., 2023) proposed a mutual information-based ranking method for natural language instructions and generated solutions. Reranking has also been approached using execution-based metrics. MBR-exec (Shi et al., 2022) minimizes a loss function across all solutions, while AlphaCode (Li et al., 2022) clusters solutions based on execution outputs. LEVER (Ni et al., 2023) uses a verifier to assess program correctness, and CodeT (Chen et al., 2023) generates high-quality test cases. Our approach stands out as it does not require model training or fine-tuning and can complement methods like LEVER (Ni et al., 2023).

8 Conclusion

We propose SRank, a novel reranking strategy designed to extract optimal code generation solutions from CodeLLMs. SRankfocuses on modeling inter-cluster relationships to identify clusters with the highest functional overlap with others. By prioritizing the cluster with the most comprehensive interaction, often indicating correct functionality, we can pinpoint the most promising solution. Incorporating these inter-cluster relationships into the ranking pipeline enhances solution identification.

We showcase the state-of-the-art performance of **SRank** on pass@1 across various well-known CodeLLMs, surpassing other ranking methods like

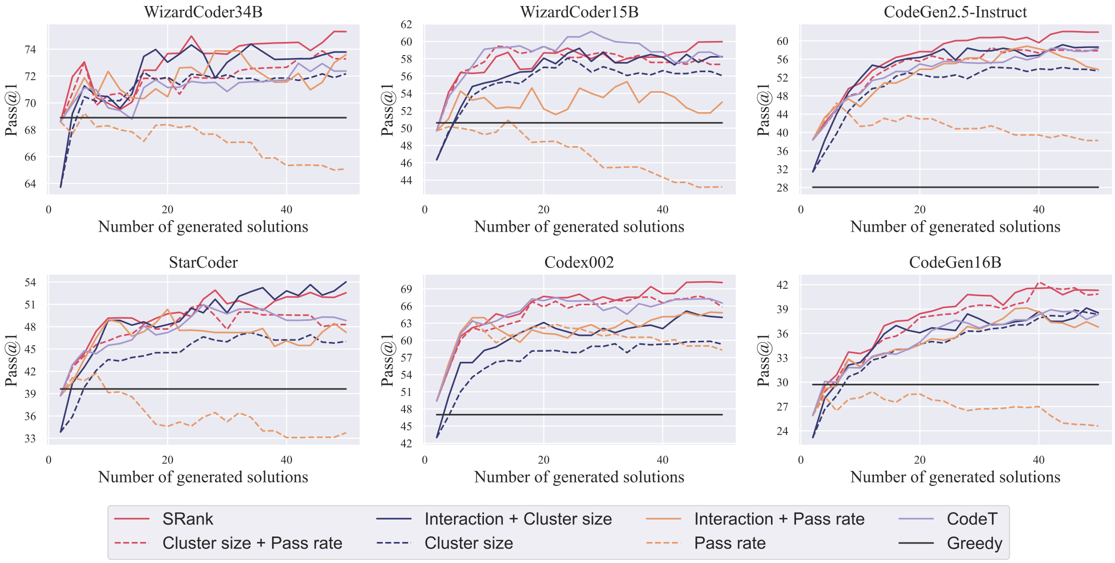


Figure 5: Ablation study on scaling number of sampled solutions vs. pass@1 on HumanEval.

CodeT and Coder-Reviewer in extensive evaluations. Our thorough analysis further highlights the method's effectiveness in realistic scenarios with a limited number of solutions and test cases. These findings are crucial as they address the challenges of code generation in real-world applications, illuminating strategies for selecting superior solutions within constrained coding environments.

Limitations

Our approach presents several opportunities for further development and refinement. While our method demonstrates superiority over others in our empirical evaluation, which focuses on Python, extending SRank to a multilingual benchmark would provide a more comprehensive assessment of its effectiveness across different programming languages.

SRank has shown promising results in its current scope, which primarily focuses on functions with clear output metrics, such as arithmetic or algorithmic problems. To extend SRank to more complex coding tasks, such as system architecture, memory management, and process management, we would need to validate that our underlying assumption of diverse incorrect solutions holds true in these domains and adapt the function δ to effectively capture the intricacies of these tasks. This presents an exciting opportunity to expand the applicability of our approach.

While implementing SRank could introduce computational overhead due to the need to generate a large number of candidate solutions and test cases, our ablation study demonstrates that SRank can achieve superior performance with a limited number of sampled solutions and test cases. To further optimize the efficiency of our method, particularly in resource-constrained environments or when rapid solution generation is crucial, we plan to conduct additional analyses and explore potential optimizations. Currently, SRank focuses on assessing functional correctness by modeling the functional overlap among solutions. While functional correctness is a critical aspect of code quality, we acknowledge the importance of other factors such as efficiency and robustness. Integrating these factors into our framework presents an exciting direction for future work, as it would enhance the comprehensiveness and robustness of SRank in meeting the diverse requirements of real-world coding tasks. By addressing these aspects, we aim to make SRank an even more valuable tool for the coding community.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. Technical report, Anthropic.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 18–30.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Martin A. Fischler and Robert C. Bolles. 1981. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *CoRR*, abs/2001.08361.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023a. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: learning to verify language-to-code generation with execution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2006–2018.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. 2024. Stable code technical report. *arXiv preprint arXiv:2404.01226*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods* *in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3533–3546. Association for Computational Linguistics.

HQ To, NDQ Bui, J Guo, and TN Nguyen. Better language models of code through self-improvement (2023). *DOI: https://doi. org/10.48550/arXiv*, 2304.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *CoRR*, abs/2303.18223.

A Validity of our assumption

A.1 Quantitative Assessment

To consolidate our assumption, we provide additional results besides Section 6. Formally, we introduce additional notations. $s_i = s_j$ is defined as the two solutions sharing the same semantic functionality, while $s_i \neq s_j$ denotes the opposite. We then compute the probability of having two equivalently semantic solutions below:

$$p(s_i = s_j) = \frac{\sum_k \binom{|C_k|}{2}}{\binom{|F|}{2}} \tag{3}$$

Given that our clustering algorithm ensures all the solutions within a cluster share functionality, and solutions from distinct clusters are certainly not equivalently semantic, we just need to randomly select two solutions from a cluster to satisfy $s_i = s_j$. Thus, the probability of having two equivalently semantic incorrect solutions is

$$\begin{aligned} &p(s_i = s_j, s_i \text{ and } s_j \text{ are incorrect}) \\ &\qquad = \frac{\sum_k \binom{|C_k|^*}{2}}{\binom{|F|}{2}} \end{aligned} \tag{4}$$

We can easily compute the probability of two wrong solutions given that they are equivalently-semantic

$$\begin{aligned} &p(s_i \text{ and } s_j \text{ are incorrect} | s_i = s_j) \\ &= \frac{p(s_i = s_j, s_i \text{ and } s_j \text{ are incorrect})}{p(s_i = s_j)} \end{aligned} \tag{5}$$

Results in Table 4 show that the probability of having two equivalently semantic incorrect solutions is pretty low at around 0.092. Given that two solutions share the same functionality, the probability for them to be wrong is approximately 0.262, far lower than that of having them be correct. These findings provide strong support for our assumption.

A.2 Qualitative Assessment

Besides quantitative evaluation supporting our assumption, we offer qualitative examples through interaction matrices of solutions generated by StarCoder on HumanEval, as shown in Figure 6. Each matrix is divided into four separate patches by two red-dashed lines: top-left, top-right, bottom-left, and bottom-right. The left region of the vertical line represents clusters, including correct solutions, while the right region encompasses incorrect clusters; similarly, the top region above the horizontal line comprises correct solutions. The diagonal lines are disregarded as their values represent the self-interaction of clusters. It is clearly seen that the top-left patches are notably brighter, whereas the bottom-right patches are virtually dark. These observations demonstrate that correct clusters interact with each other comprehensively, indicating a high probability of functional overlap between correct solutions, while the opposite holds true for incorrect solutions.

Model	$p(s_i = s_j)$	$p(s_i = s_j, s_i$ and s_j are incorrect)	$p(s_i$ and s_j are incorrect$\|s_i = s_j)$
StarCoder	0.2176	0.0559	0.2569
CodeGen2.5-Instruct	0.2692	0.0737	0.2738
WizardCoder15B	0.4596	0.1134	0.2467
WizardCoder34B	0.5689	0.0990	0.1740
Codex002	0.3218	0.0807	0.2508
CodeGen16B	0.3482	0.1279	0.3673

Table 4: Probabilities of two equivalently-semantic solutions and two equivalently-semantic incorrect solutions among different models

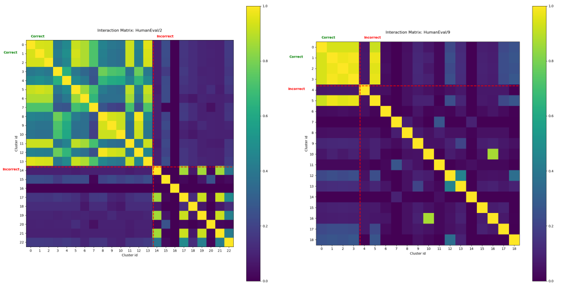


Figure 6: Demonstration of our assumption shows there is a low functional agreement among incorrect solutions. The two graphs represent two problems chosen from Human-Eval, with solutions generated by StarCoder.

B Generation Prompts

In this section, we list all the prompts feeding to CodeLLMs for generation process of code solutions and test cases. For concrete demonstration, we use the same problem 'HumanEval/8' throughout this section.

B.1 Code generation

Demonstrated prompts for code generation includes Figure 7, Figure 8, Figure 9, and Figure 10.

B.2 Test Generation

We give example test case generation prompts in Figure 11, Figure 12, Figure 13, and Figure 14.

B.3 CoderReviewer Prompts

In CoderReviewer, Coder generates code solution given the task description and Reviewer checks the generated solution by measuring the sequence probability of the description given the solution.

For generating code solutions, we use the same prompt as in the Code Generation section. For Reviewer, to adapt to instruction-tuned models, here we list the prompt we manually design to calculate the probability of task description given the code implementation with respect to CodeLLMs.

```
Below is an instruction that describes a
    task. Write a response that
    approriately completes the requests.

### Instruction:
Create a Python script for this problem:
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""

### Response:
```

Figure 7: Code generation prompt used with WizardCoder34B, WizardCoder15B, and CodeGen2.5-Instruct

```
# Here is the correct implementation of
    the code exercise
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```

Figure 8: Code generation prompt used with StarCoder

```
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```

Figure 9: Code generation prompt used with Codex002, CodeGen16B

~~~
Complete the following Python script. We
     will be using the output you
    provide as-is to create new files,
    so please be precise and do not
    include any other text. Your output
    needs to be ONE file; Otherwise, it
    will break the system. Moreover,
    your response must include the
    provided code and the newly
    generated code. Your output must
    consist ONLY of the language and
    code, in the fenced code block
    format:
```language
CODE
~~~
Here is the initial code:
```python
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```
```

Figure 10: Code generation prompt used with Claude 3 Opus

```
Below is an instruction that describes a
     task. Write a response that
    appropriately completes the request.

### Instruction:
I have this function stub, please
    generate 50 test cases for this
    function. The function stub is as
    follow:
```python
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
    pass
```
- Each test case is in the form of
    assertion statement, for example:
    assert sum_product(...) == ...
- Each test case is in a single line
- The length of each test case should be
     too long, ideally less than or
    equal to 150 letters
- The test input should not be too long
- The inputs of test cases should be
    diverse and cover corner cases of
    the function
- Test cases should not be repeated

### Response: Here are 50 test cases for
     function `sum_product`:
assert sum_product(
```

Figure 11: Test case generation prompt used with WizardCoder34B, WizardCoder15B, and CodeGen2.5-Instruct

```
<filename>solutions/solution_1.py
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
    pass

# check the correctness of sum_product
assert sum_product(
```

Figure 12: Test case generation prompt used with StarCoder
```

```
def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
    pass

# check the correctness of sum_product
assert sum_product(
```

Figure 13: Test generation prompt used with Codex002, CodeGen16B

~~~
You are given a function stub. Your task
     is to generate 20 test cases for
    this function. Each test case must
    be in the form of a single-line
    assertion statement in Python. For
    example: `assert function_name(input
    ) == output`. Replace `function_name
    ` with the actual function name from
     the stub, and replace `input` and `
    output` with the appropriate values.
     The test inputs must be diverse and
     cover all possible edge cases of
    the function. Do not repeat any test
     case. Additionally, each test case
    input must not be too
    computationally expensive when
    executing the function. Your output
    must consist ONLY of the test cases,
     with each test case on a new line.
    Do not include any other text.

The function stub is as follows:
```python
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```
~~~

Figure 14: Test generation prompt used with Claude 3 Opus

```
### Instruction:
Write a docstring for the above function
    :
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    product = 1
    for number in numbers:
        product *= number
    return (sum(numbers), product)

### Response: Here's the docstring for
    the above function
def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```

Figure 15: Prompt used for measuring sequence probability of task description given a specific code implementation of models WizardCoder34B, WizardCoder15B, and CodeGen2.5-Instruct.

The prompt templates we used including Figure 15, Figure 16, and Figure 17.

C Scaling Number of Generated Test Cases and Sampled Solutions on MBPP-S

This section complements the results on MBPP-S dataset from Section Scaling Number of Generated Test Cases and Section Scaling Number of Sampled Solutions. The setting for MBPP-S is the same as HumanEval mentioned in each above section. Figure 18 and Figure 19 shows pass@1 as function of the number of generated test cases and sampled solutions respectively on MBPP-S benchmark.

D CoderReviewer Ranking Criteria as Cluster Features

We investigate adding cluster interaction with CoderReviewer ranking criteria as cluster features, where Coder represents the likelihood $P_{CodeLLM}(x|y)$ and Reviewer represents $P_{CodeLLM}(y|x)$. CoderReviewer is the multiplication $P_{CodeLLM}(x|y)P_{CodeLLM}(x|y)$, with x as the task description and y as generated code solution. The prefixed N. before each term indicates that the likelihood is normalized by the number of sequence tokens, which was shown

```
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    product = 1
    for number in numbers:
        product *= number
    return (sum(numbers), product)

# Write a docstring for the above
    function
def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```

Figure 16: Prompt used for measuring sequence probability of task description given a specific code implementation of model StarCoder

```
from typing import List, Tuple

def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    product = 1
    for number in numbers:
        product *= number
    return (sum(numbers), product)

# write a docstring for the above
    function
def sum_product(numbers: List[int]) ->
    Tuple[int, int]:
    """ For a given list of integers,
        return a tuple consisting of a
        sum and a product of all the
        integers in a list. Empty sum
        should be equal to 0 and empty
        product should be equal to 1."""
```

Figure 17: Prompt used for measuring sequence probability of task description given a specific code implementation of models Codex002, CodeGen16B

	HumanEval	MBPP-S
N.Coder	57.15	56.73
Interaction + N.Coder	62.69	59.50
N.Reviewer	55.33	55.41
Interaction + N.Reviewer	62.69	59.74
N.CoderReviewer	62.67	59.39
Interaction + N.CoderReviewer	**63.30**	**61.38**

Table 5: CoderReviewer ranking criterions as cluster fearures.

to improve performance in previous work. The cluster feature in these cases is the criteria score of the solution with maximum score within that cluster.

Experimental results in Table 5 show consistent improvement when adding cluster interaction over CoderReviewer. This demonstrates that when the interaction matrix is combined with CoderReviewer, the interaction matrix can improve CoderReviewer's performance even further, indicating that our method is adaptable to different reranking methods.

E Reranking with Interaction Only

To further validate that modeling inter-cluster interactions aids in ranking clusters, we present the results of reranking solely by matrix I without any cluster features V in Table 6. In this scenario, V is a column vector with 1 at every entry. Consequently, the ith entry in R is the sum of functional overlap between the ith cluster and all other clusters.

The results demonstrate that incorporating interaction modeling consistently elevated performance beyond random. Furthermore, reranking with interaction alone significantly outperforms the performance of greedy decoding in some cases, such as 51.13 vs 28.49 for CodeGen2.5, 50.01 vs 39.63 for StarCoder, 62.75 vs 47.00 for CodeX002 in HumanEval. This result futher confirms the effectiveness of our method **SRank**.

F Additional Results with Closed-Source LLM

In the previous section, we presented code generation ranking results with both open-source and closed-source models (Codex002), which is not the strongest and up-to-date model for code. To examine our method on contemporary and high-capability LLMs, we selected Anthropic Claude 3

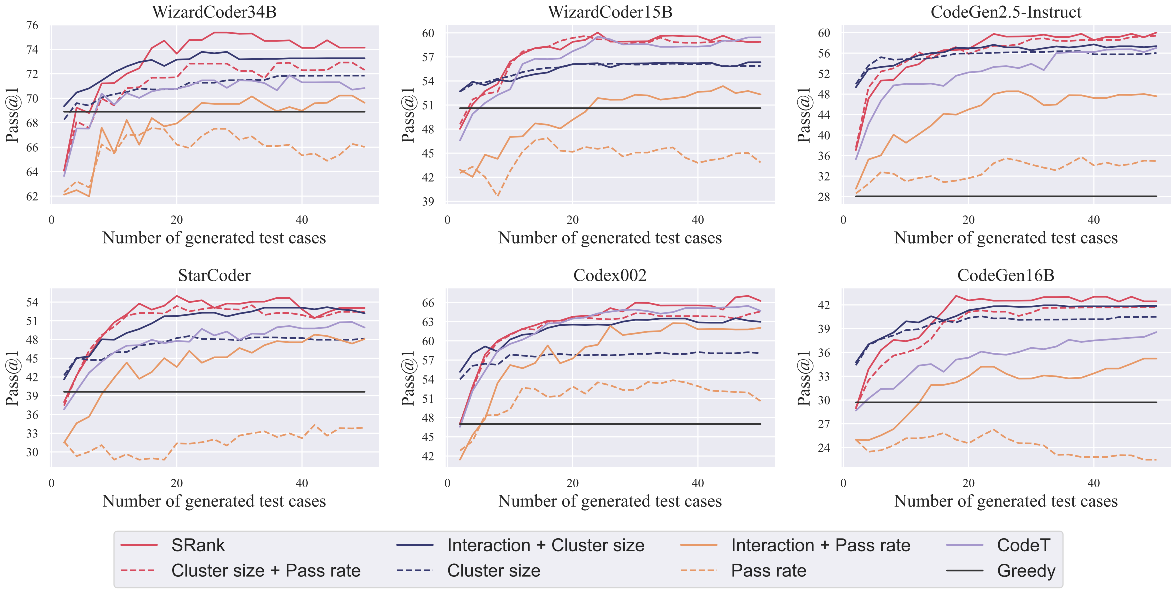


Figure 18: Ablation study on scaling number of model generated test cases vs. pass@1 on MBPP-S.

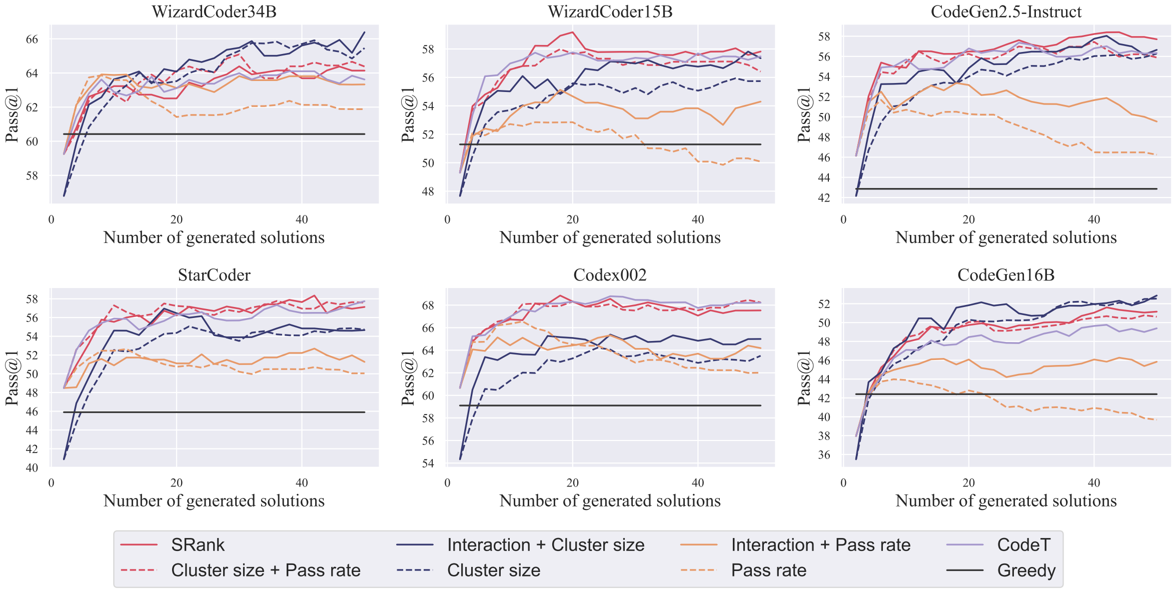


Figure 19: Ablation study on scaling number of sampled solutions vs. pass@1 on MBPP-S.

	HumanEval					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Greedy	**68.90**	**50.61**	28.05	39.63	47.00	29.70
Random	59.88	45.20	26.68	32.55	37.06	22.78
Interaction Only	66.49	49.79	**51.13**	**50.01**	**62.75**	**31.31**
	MBPP-S					
	WizardCoder34B	WizardCoder15B	CodeGen2.5-Instruct	StarCoder	Codex002	CodeGen16B
Greedy	**60.42**	51.29	42.86	45.90	**58.10**	**42.40**
Random	54.37	45.72	34.60	39.26	47.50	31.54
Interaction Only	59.60	**53.94**	**47.70**	**48.85**	57.49	41.98

Table 6: Results of pass@1 on HumanEval and MBPP-S benchmarks when reranking merely by interaction among cluster without any cluster feature.

Method	Claude 3 Opus
Random	78.03
Greedy	78.05
CodeT	77.42
SRank	**79.18**

Table 7: Results of pass@1 on HumanEval in the zero-shot setting of closed-source model Claude 3 Opus with different LLM decoding strategies and reranking methods.

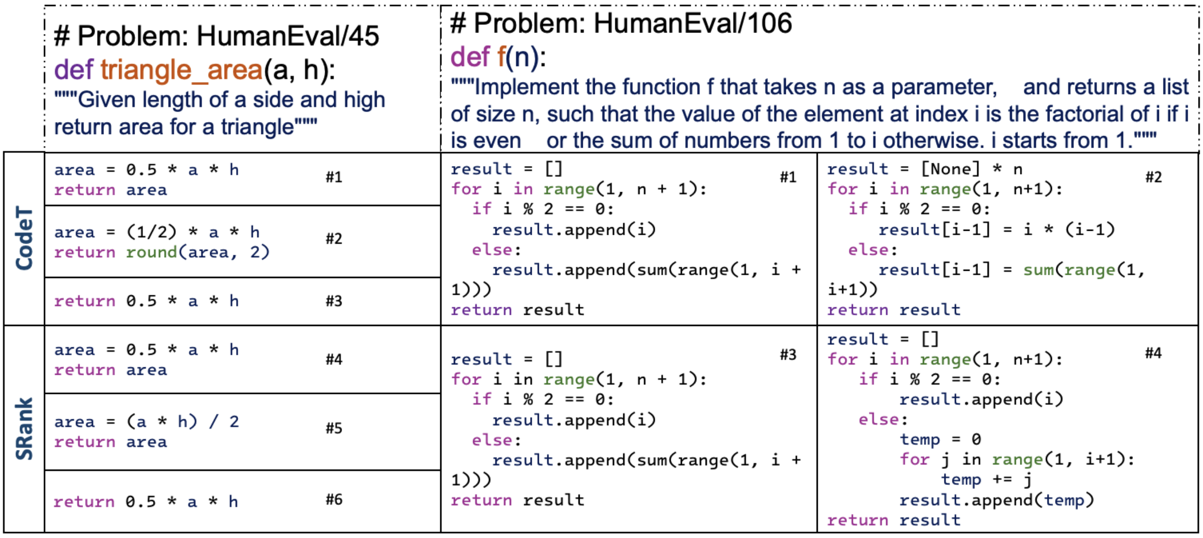
Opus (Anthropic, 2024) as a representative model for our experiments. Table 7 shows the results of code generation on HumanEval using different decoding strategies, such as greedy search and random sampling, as well as when reranking methods like SRankand CodeT are employed. Even with such a large and powerful LLM, SRankstill achieves impressive performance compared to other baselines, while CodeT degrades performance of baselines.

In this experiment, we followed similar hyperparameter settings as with other models, setting the temperature to 0.8 and top-p to 0.95. The results indicate that random sampling and greedy search yield roughly the same performance, suggesting that the model is very confident in its responses and the randomness in output sequences is minimal. This behavior is further confirmed by observing the generated code solutions and test cases, where there is little variance between the samples and a higher rate of duplication compared to other models we experimented with. For such large and high-capability models, the benefits from reranking strategies like SRankare most fully realized when sufficient randomness is introduced in the sampling process by adjusting hyperparameters, such as temperature, top-k, and top-p.

Notably, the highest reported score for Claude 3 Opus from Anthropic on HumanEval is 84.9, which is higher than the numbers presented in this work. The explanation for this discrepancy is that the test cases included in the HumanEval prompt were excluded in our settings, following previous work in code generation reranking.

G Case Study

In this section, we aim to compare the quality of cluster samples from CodeT and our method. The top 1 clusters ranked by CodeT and SRank are shown in Figure 20. It is clear that functional inconsistency among solutions plagues CodeT's clusters. For example, in the problem HumanEval 45, solution #2 is not semantically equivalent to solutions #1 and #3, and in the problem HumanEval 106, solution #1 differs from solution #2. This phenomenon is explained by the fact that CodeT groups solutions that pass the same set of test cases into clusters. As a result, when computing the pass@k, inconsistency in functionalities among top cluster solutions can degrade performance. *SRank*, on the other hand, considers execution outputs for the clustering process, ensuring functional consistency among solutions within a cluster.



```
# Problem: HumanEval/45
def triangle_area(a, h):
"""Given length of a side and high
return area for a triangle"""
```

CodeT

```
area = 0.5 * a * h
return area
```

#1

```
area = (1/2) * a * h
return round(area, 2)
```

#2

```
return 0.5 * a * h
```

#3

SRank

```
area = 0.5 * a * h
return area
```

#4

```
area = (a * h) / 2
return area
```

#5

```
return 0.5 * a * h
```

#6

```
# Problem: HumanEval/106
def f(n):
"""Implement the function f that takes n as a parameter,   and returns a list
of size n, such that the value of the element at index i is the factorial of i if i
is even    or the sum of numbers from 1 to i otherwise. i starts from 1."""
```

CodeT

```
result = []
for i in range(1, n + 1):
  if i % 2 == 0:
    result.append(i)
  else:
    result.append(sum(range(1, i +
1)))
return result
```

#1

```
result = [None] * n
for i in range(1, n+1):
  if i % 2 == 0:
     result[i-1] = i * (i-1)
  else:
     result[i-1] = sum(range(1,
i+1))
return result
```

#2

SRank

```
result = []
for i in range(1, n + 1):
  if i % 2 == 0:
    result.append(i)
  else:
    result.append(sum(range(1, i +
1)))
return result
```

#3

```
result = []
for i in range(1, n+1):
   if i % 2 == 0:
       result.append(i)
   else:
       temp = 0
       for j in range(1, i+1):
           temp += j
       result.append(temp)
return result
```

#4

Figure 20: Case studies from HumanEval's problems with the highest-score clusters produced by CodeT versus SRank using CodeGen2.5-Instruct.